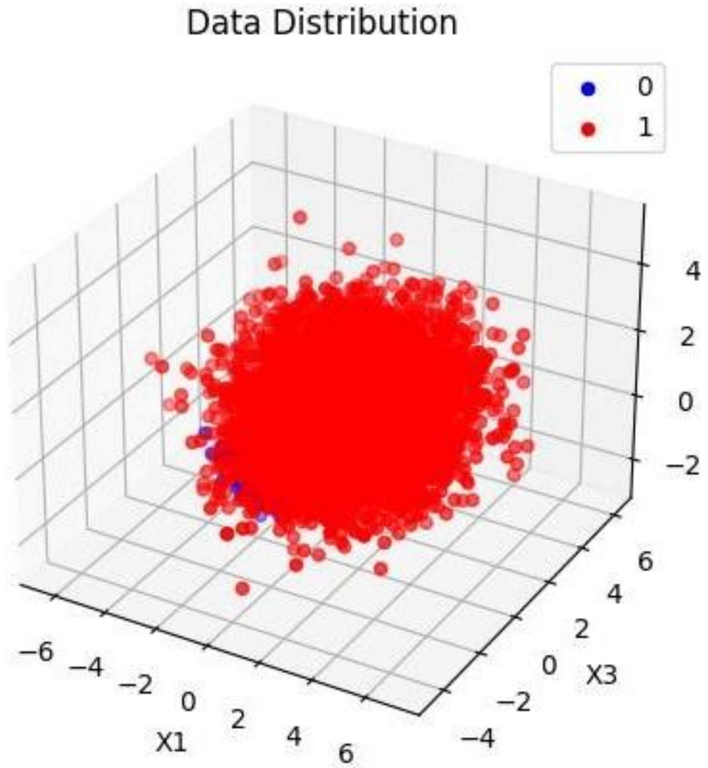
ERM_Classifier

A.



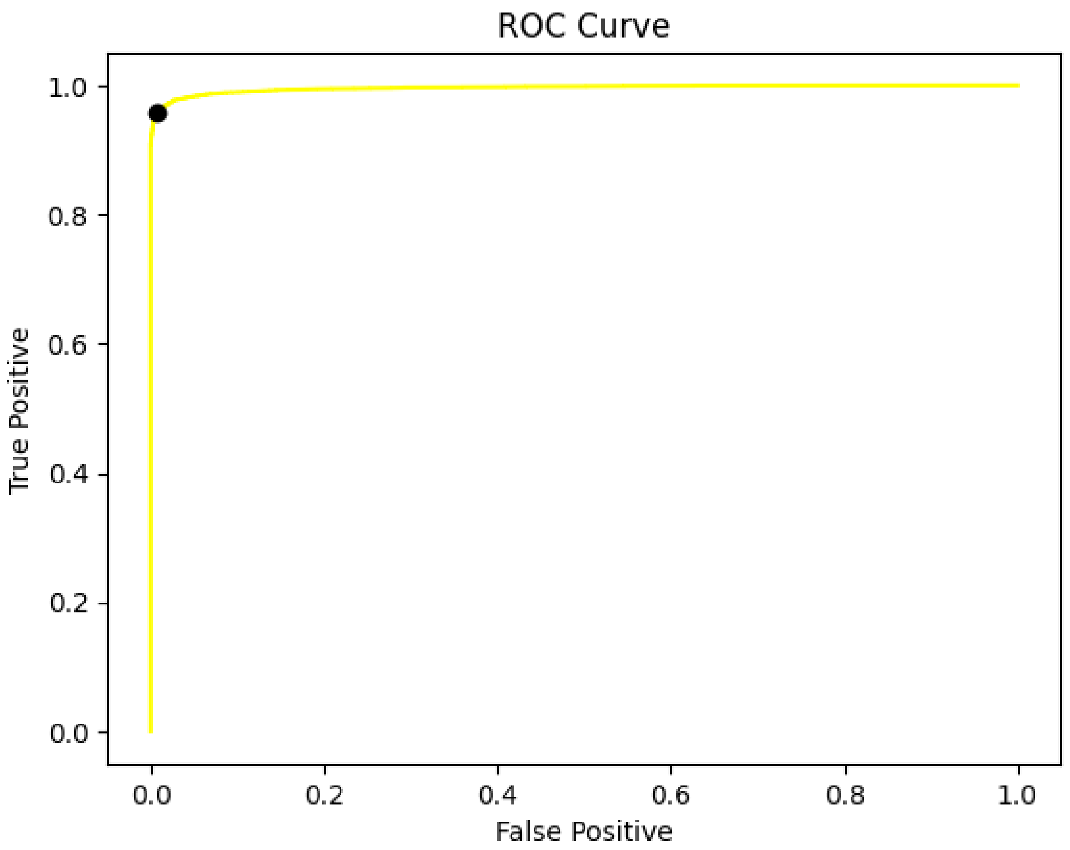
1. Minimum Expected Risk Classification Rule is as follows:

$$\frac{g(x|m_0, C_0)}{g(x|m_1, C_1)} \gtrless \frac{P(L=0)(\lambda_{10}-\lambda_{00})}{P(L=1)(\lambda_{01}-\lambda_{11})} = \frac{(0.65)(\lambda_{10}-\lambda_{00})}{(0.35)(\lambda_{01}-\lambda_{11})}$$

$$\frac{g(x|m_0, C_0)}{g(x|m_1, C_1)} \underset{(D=0)}{\overset{(D=1)}{\gtrless}} \frac{0.65}{0.35}\frac{(\lambda_{10}-\lambda_{00})}{(\lambda_{01}-\lambda_{11})} = \gamma$$

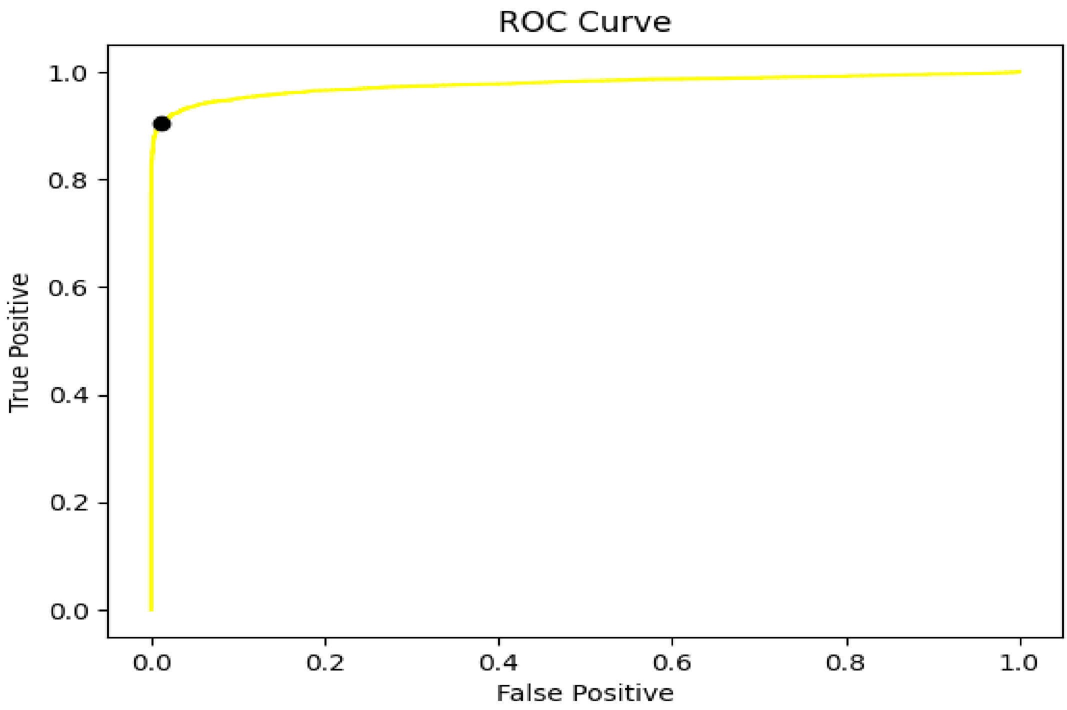
$$\gamma = 1.8571$$

2. The ROC curve is generated by approximating the variation of threshold value γ from 0 to ∞. Practically, γ values are sorted and the mid-points between consecutive two values are considered as threshold points ranging from minimum to maximum.
3. Theoretically, the value of γ is found out by dividing the class priors (0.65/0.35) which is equal to 1.85714. With this threshold, false positives and true positives are computed which help in estimating the Minimum P(error). It can be observed that there is negligible difference in the theoretical and experimental values.



Value of Gamma (practical) is 1.86554
Corresponding minimum error is 0.01961
Value of Gamma (Ideal) is 1.85714
Corresponding minimum error is 0.01961

B.



Value of Gamma (practical) is 0.42374
Corresponding minimum error is 0.04101
Value of Gamma (Ideal) is 1.85714
Corresponding minimum error is 0.05938

1. Minimum Expected Risk Classification Rule is as follows:

$$\frac{g(x|m_0, C_0)}{g(x|m_1, C_1)} \gtrless \frac{P(L=0)(\lambda_{10}-\lambda_{00})}{P(L=1)(\lambda_{01}-\lambda_{11})} = \frac{(0.65)(\lambda_{10}-\lambda_{00})}{(0.35)(\lambda_{01}-\lambda_{11})}$$

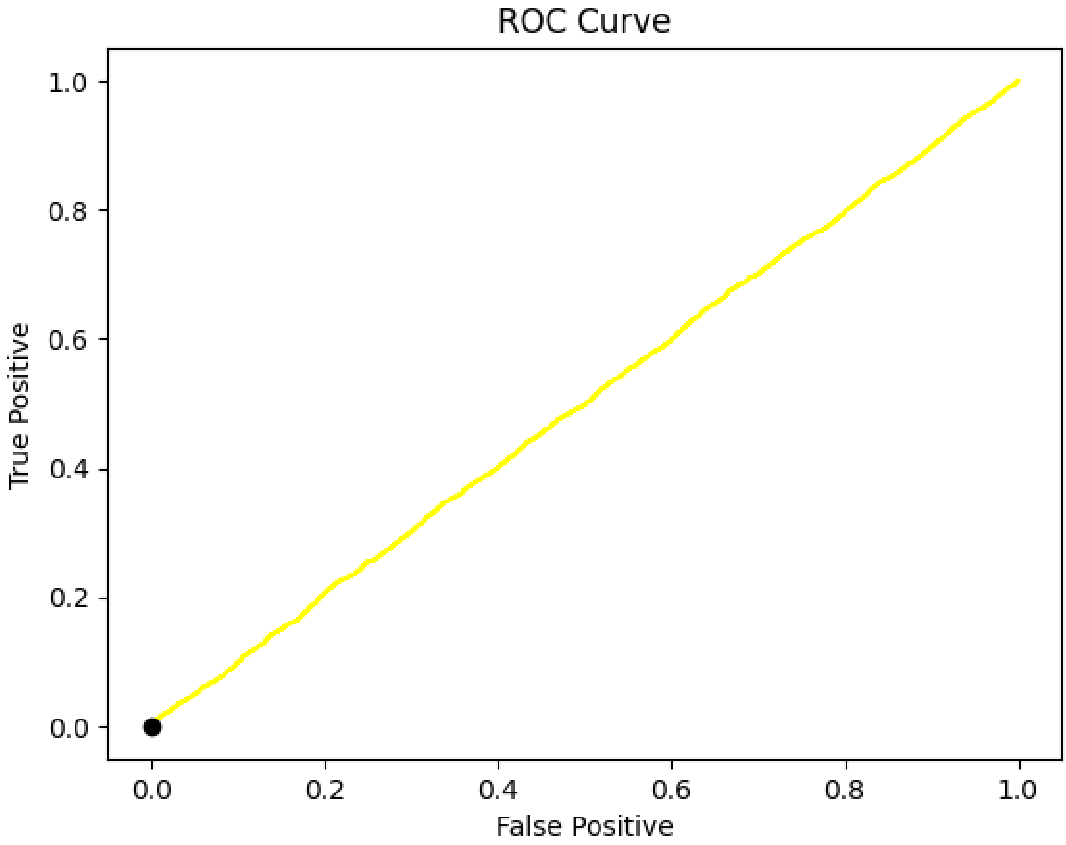
$$\frac{g(x|m_0, C_0)}{g(x|m_1, C_1)} \underset{(D=0)}{\overset{(D=1)}{\gtrless}} \frac{0.65}{0.35}\frac{(\lambda_{10}-\lambda_{00})}{(\lambda_{01}-\lambda_{11})} = \gamma$$

$$\gamma = 1.8571$$

C.

1. LDA Classification Rule is as follows:

(D = 1) transpose(w_ LDA) X > τ (D = 0)



Value of Tau (practical) is 4.50229
Corresponding minimum error is 0.35018
Value of Tau (Ideal) is 1.85714
Corresponding minimum error is 0.42054

Please find the below GitHub Link for the code:

GitHub Link

Appendix:
A.

```
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
np.random.seed(10)

def generate_class_labels(p):
    class_labels = np.random.rand(sample_size)
    class_labels = class_labels >= p[0]
    return class_labels.astype(int)

def generate_samples(sample_size, class_labels):
    # generate samples for each class
    samples = np.zeros(shape = [sample_size, features])
    for index in range(sample_size):
        if class_labels[index] == 0:
            samples[index, :] = np.random.multivariate_normal(mean_0, covariance_0)
        elif class_labels[index] == 1:
            samples[index, :] = np.random.multivariate_normal(mean_1, covariance_1)
    return samples

def discriminant_score(samples, mean_0, mean_1, covariance_0, covariance_1):
    gaussian_pdf_0 = np.log(multivariate_normal.pdf(samples, mean_0, covariance_0))
    gaussian_pdf_1 = np.log(multivariate_normal.pdf(samples, mean_1, covariance_1))
    return gaussian_pdf_1 - gaussian_pdf_0

def calculate_mid_points(d_score_sorted):
    threshold = []
    for i in range( len(d_score_sorted) - 1 ):
        threshold.append( (d_score_sorted[i] + d_score_sorted[i+1]) / 2.0 )
    threshold = np.array(threshold)
    return threshold

def classify(d_score, threshold, p):
    true_positive = [0] * len(threshold)
    false_positive = [0] * len(threshold)
    error = [0] * len(threshold)
    for (index, th) in enumerate(threshold):
        decision = (d_score >= th)
        true_positive[index] = (np.size(np.where((decision == 1) & (class_labels == 1))))
        true_positive[index] = true_positive[index] / np.size(np.where(class_labels == 1))
        false_positive[index] = (np.size(np.where((decision == 1) & (class_labels == 0))))
        false_positive[index] = false_positive[index] / np.size(np.where(class_labels == 0))
        error[index] = (p[1] * false_positive[index]) + (p[0] * (1 - true_positive[index]))
    return true_positive, false_positive, error

# Number of samples
sample_size = 10000
features = 4
# class-conditional Gaussian pdf parameters
mean_0 = np.array([-1/2, -1/2, -1/2, -1/2])
covariance_0 = np.array([[2/4, -0.5/4, 0.3/4, 0], [-0.5/4, 1/4, -0.5/4, 0], [0.3/4, -0.5/4, 1/4, 0], [0, 0,
```

0, 2/4]])

```
mean_1 = np.array([1, 1, 1, 1])
covariance_1 = np.array([[1, 0.3, -0.2, 0], [0.3, 2, 0.3, 0], [-0.2, 0.3, 1, 0], [0, 0, 0, 3]])
# class priors
p = [0.35, 0.65]

# generating class labels for the given number of samples
class_labels = generate_class_labels(p)

# generating the samples / data
samples = generate_samples(sample_size, class_labels)

#Plot Data / Samples
fig = plt.figure()
ax = plt.axes(projection = "3d")
samples_class0 = ax.scatter(samples[(class_labels==0),3], samples[(class_labels==0),1],
samples[(class_labels==0),2],'+',color ='blue', label="0")
samples_Class1 =
ax.scatter(samples[(class_labels==1),3],samples[class_labels==1,1],samples[class_labels==1,2],'.',c
olor = 'red', label="1")
plt.xlabel('X1')
plt.ylabel('X3')
ax.set_zlabel('X2')
ax.legend()
plt.title('Data Distribution')
plt.show()

# calculate discriminant score
d_score = discriminant_score(samples, mean_0, mean_1, covariance_0, covariance_1)

d_score_sorted = np.sort(d_score)
# calculating the threshold values (mid - points of d_score)
threshold = calculate_mid_points(d_score_sorted)

# classifying the data and calculating the p error
true_positive, false_positive, error = classify(d_score, threshold, p)
# Plot ROC curve
plt.plot(false_positive, true_positive, color = 'yellow')
plt.xlabel('False Positive')
plt.ylabel('True Positive')
plt.title('ROC Curve')
plt.plot(false_positive[np.argmin(error)], true_positive[np.argmin(error)],'o',color = 'black')
plt.show()
print(f'Value of Gamma (practical) is {round(np.exp(threshold[np.argmin(error)]), 5)} ')
print(f'Corresponding minimum error is {round(np.min(error), 5)}')

# computing idea value
ideal_gamma = p[1] / p[0]
ideal_threshold = np.log(ideal_gamma)
ideal_decision = (d_score >= ideal_threshold)
ideal_true_positive = (np.size(np.where((ideal_decision == 1) & (class_labels ==
1))))/np.size(np.where(class_labels == 1))
ideal_false_positive = (np.size(np.where((ideal_decision == 1) & (class_labels ==
0))))/np.size(np.where(class_labels == 0))
ideal_error = (p[1] * ideal_false_positive) + (p[0] * (1 - ideal_true_positive))
print(f'Value of Gamma (Ideal) is {round(ideal_gamma, 5)} ')
```

```
print(f'Corresponding minimum error is {round(ideal_error, 5)}')
```

B

```
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
np.random.seed(10)

def generate_class_labels(p):
    class_labels = np.random.rand(sample_size)
    class_labels = class_labels >= p[0]
    return class_labels.astype(int)

def generate_samples(sample_size, class_labels):
    # generate samples for each class
    samples = np.zeros(shape = [sample_size, features])
    for index in range(sample_size):
        if class_labels[index] == 0:
            samples[index, :] = np.random.multivariate_normal(mean_0, covariance_0)
        elif class_labels[index] == 1:
            samples[index, :] = np.random.multivariate_normal(mean_1, covariance_1)
    return samples

def discriminant_score(samples, mean_0, mean_1, features):
    gaussian_pdf_0 = np.log(multivariate_normal.pdf(samples, mean_0, np.eye(features, features)))
    gaussian_pdf_1 = np.log(multivariate_normal.pdf(samples, mean_1, np.eye(features, features)))
    return gaussian_pdf_1 - gaussian_pdf_0

def calculate_mid_points(d_score_sorted):
    threshold = []
    for i in range( len(d_score_sorted) - 1 ):
        threshold.append( (d_score_sorted[i] + d_score_sorted[i+1]) / 2.0 )
    threshold = np.array(threshold)
    return threshold

def classify(d_score, threshold, p):
    true_positive = [0] * len(threshold)
    false_positive = [0] * len(threshold)
    error = [0] * len(threshold)
    for (index, th) in enumerate(threshold):
        decision = (d_score >= th)
        true_positive[index] = (np.size(np.where((decision == 1) & (class_labels == 1))))
        true_positive[index] = true_positive[index] / np.size(np.where(class_labels == 1))
        false_positive[index] = (np.size(np.where((decision == 1) & (class_labels == 0))))
        false_positive[index] = false_positive[index] / np.size(np.where(class_labels == 0))
        error[index] = (p[1] * false_positive[index]) + (p[0] * (1 - true_positive[index]))
    return true_positive, false_positive, error

# Number of samples
sample_size = 10000
features = 4
# class-conditional Gaussian pdf parameters
mean_0 = np.array([-1/2, -1/2, -1/2, -1/2])
```

```
covariance_0 = np.array([[2/4, -0.5/4, 0.3/4, 0], [-0.5/4, 1/4, -0.5/4, 0], [0.3/4, -0.5/4, 1/4, 0], [0, 0, 0, 2/4]])
mean_1 = np.array([1, 1, 1, 1])
covariance_1 = np.array([[1, 0.3, -0.2, 0], [0.3, 2, 0.3, 0], [-0.2, 0.3, 1, 0], [0, 0, 0, 3]])
# class priors
p = [0.35, 0.65]

# generating class labels for the given number of samples
class_labels = generate_class_labels(p)

# generating the samples / data
samples = generate_samples(sample_size, class_labels)

#Plot Data / Samples
fig = plt.figure()
ax = plt.axes(projection = "3d")
samples_class0 = ax.scatter(samples[(class_labels==0),3], samples[(class_labels==0),1],
samples[(class_labels==0),2],'+',color ='blue', label="0")
samples_Class1 =
ax.scatter(samples[(class_labels==1),3],samples[class_labels==1,1],samples[class_labels==1,2],'.',color = 'red', label="1")
plt.xlabel('X1')
plt.ylabel('X3')
ax.set_zlabel('X2')
ax.legend()
plt.title('Data Distribution')
plt.show()

# calculate discriminant score
d_score = discriminant_score(samples, mean_0, mean_1, features)

d_score_sorted = np.sort(d_score)
# calculating the threshold values (mid - points of d_score)
threshold = calculate_mid_points(d_score_sorted)

# classifying the data and calculating the p error
true_positive, false_positive, error = classify(d_score, threshold, p)
# Plot ROC curve
plt.plot(false_positive, true_positive, color = 'yellow')
plt.xlabel('False Positive')
plt.ylabel('True Positive')
plt.title('ROC Curve')
plt.plot(false_positive[np.argmin(error)], true_positive[np.argmin(error)],'o',color = 'black')
plt.show()
print(f'Value of Gamma (practical) is {round(np.exp(threshold[np.argmin(error)]), 5)} ')
print(f'Corresponding minimum error is {round(np.min(error), 5)}')

# computing idea value
ideal_gamma = p[1] / p[0]
ideal_threshold = np.log(ideal_gamma)
ideal_decision = (d_score >= ideal_threshold)
ideal_true_positive = (np.size(np.where((ideal_decision == 1) & (class_labels ==
1))))/np.size(np.where(class_labels == 1))
ideal_false_positive = (np.size(np.where((ideal_decision == 1) & (class_labels ==
0))))/np.size(np.where(class_labels == 0))
```

```
ideal_error = (p[1] * ideal_false_positive) + (p[0] * (1 - ideal_true_positive))
print(f'Value of Gamma (Ideal) is {round(ideal_gamma, 5)} ')
print(f'Corresponding minimum error is {round(ideal_error, 5)}')
```

C

```
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from numpy import linalg as linA
np.random.seed(10)

def generate_class_labels(p):
    class_labels = np.random.rand(sample_size)
    class_labels = class_labels >= p[0]
    return class_labels.astype(int)

def generate_samples(sample_size, class_labels):
    # generate samples for each class
    samples = np.zeros(shape = [sample_size, features])
    for index in range(sample_size):
        if class_labels[index] == 0:
            samples[index, :] = np.random.multivariate_normal(mean_0, covariance_0)
        elif class_labels[index] == 1:
            samples[index, :] = np.random.multivariate_normal(mean_1, covariance_1)
    return samples

def calculate_mid_points(y_sorted):
    threshold = []
    for i in range( len(y_sorted) - 1 ):
        threshold.append( (y_sorted[i] + y_sorted[i+1]) / 2.0 )
    threshold = np.array(threshold)
    return threshold

def classify(y, threshold, p):
    true_positive = [0] * len(threshold)
    false_positive = [0] * len(threshold)
    error = [0] * len(threshold)
    for (index, th) in enumerate(threshold):
        decision = (y >= th)
        true_positive[index] = (np.size(np.where((decision == 1) & (class_labels == 1))))
        true_positive[index] = true_positive[index] / np.size(np.where(class_labels == 1))
        false_positive[index] = (np.size(np.where((decision == 1) & (class_labels == 0))))
        false_positive[index] = false_positive[index] / np.size(np.where(class_labels == 0))
        error[index] = (p[1] * false_positive[index]) + (p[0] * (1 - true_positive[index]))
    return true_positive, false_positive, error

def between_slass_SB(mean_0, mean_1):
    return ( (mean_0 - mean_1) * (np.transpose(mean_0 - mean_1)) )

def within_class_SW(covariance_0, covariance_1):
    return ( covariance_0 + covariance_1 )
```

```
# Data projection using LDA
def data_projection(class0, class1, w_max):
    y0 = [0] * len(class0)
    y1 = [0] * len(class1)
    y0 = np.dot(np.transpose(w_max), np.transpose(class0))
    y1 = np.dot(np.transpose(w_max), np.transpose(class1))
    y =  np.concatenate( [y0, y1] )
    return y0, y1, y

# Number of samples
sample_size = 10000
features = 4
# class-conditional Gaussian pdf parameters
mean_0 = np.array([-1/2, -1/2, -1/2, -1/2])
covariance_0 = np.array([[2/4, -0.5/4, 0.3/4, 0], [-0.5/4, 1/4, -0.5/4, 0], [0.3/4, -0.5/4, 1/4, 0], [0, 0,
0, 2/4]])
mean_1 = np.array([1, 1, 1, 1])
covariance_1 = np.array([[1, 0.3, -0.2, 0], [0.3, 2, 0.3, 0], [-0.2, 0.3, 1, 0], [0, 0, 0, 3]])
# class priors
p = [0.35, 0.65]

#Compute scatter matrix within class
s_w = within_class_SW(covariance_0, covariance_1)

#Computer scatter matrix between class
s_b = between_slass_SB(mean_0, mean_1)

# generating class labels for the given number of samples
class_labels = generate_class_labels(p)

# generating the samples / data
samples = generate_samples(sample_size, class_labels)

#Computer Eigen Values and Eigen Vectors
v, w = linA.eig( linA.inv(s_w) * s_b )

# Maximum optimization objective
w_max = w[np.argmax(v)]
class0 = samples[ np.where(class_labels == 0) ]
class1 = samples[ np.where(class_labels == 1) ]

# Data projection using LDA
y0, y1, y = data_projection(class0, class1, w_max)

y_sorted = np.sort(y)

# calculating the threshold values (mid - points of d_score)
threshold = calculate_mid_points(y_sorted)

#Plot Data / Samples
fig = plt.figure()
ax = plt.axes(projection = "3d")
samples_class0 = ax.scatter(samples[(class_labels==0),3], samples[(class_labels==0),1],
samples[(class_labels==0),2],'+',color ='blue', label="0")
```

```
samples_Class1 =
ax.scatter(samples[(class_labels==1),3],samples[class_labels==1,1],samples[class_labels
==1,2],'.',c olor = 'red', label="1")
plt.xlabel('X1')
plt.yla
bel('X3
')
ax.set_
zlabel('
X2')
ax.lege
nd()
plt.title('Data
Distribution')
plt.show()

# classifying the data and calculating the p error
true_positive, false_positive, error = classify(y,
threshold, p)

# Plot ROC curve
plt.plot(false_positive, true_positive, color
= 'yellow')plt.xlabel('False Positive')
plt.ylabel('Tru
e Positive')
plt.title('ROC
Curve')
plt.plot(false_positive[np.argmin(error)],
true_positive[np.argmin(error)],'o',color = 'black')plt.show()
print(f'Value of Tau (practical) is
{round(threshold[np.argmin(error)], 5)} ')print(f'Corresponding
minimum error is {round(np.min(error), 5)}')

# computing
idea value
ideal_gamma
= p[1] / p[0]
ideal_threshold =
np.log(ideal_gamma)
ideal_decision = (y >=
ideal_threshold)
ideal_true_positive = (np.size(np.where((ideal_decision == 1) &
(class_labels ==1))))/np.size(np.where(class_labels == 1))
ideal_false_positive = (np.size(np.where((ideal_decision == 1) &
(class_labels ==0))))/np.size(np.where(class_labels == 0))
ideal_error = (p[1] * ideal_false_positive) + (p[0] * (1 -
ideal_true_positive))print(f'Value of Tau (Ideal) is
{round(ideal_gamma, 5)} ') print(f'Corresponding minimum error
is {round(ideal_error, 5)}')
```